Chapter 5. Working with Complex Data Types

In the previous chapter, we created RPC-style services in Java. Those services dealt only with simple data types. Simple data types may suffice in many situations, but you'll eventually need to use more complex data types, like those you're familiar with in your day-to-day Java programming. In this chapter we'll look at creating services with method parameters and return values that are arrays and Java beans. We'll hold off on custom data types until the next chapter, since it's possible that any custom types you create would use the complex data types we'll be discussing here.

5.1 Passing Arrays as Parameters

Let's face it — arrays are probably the most common complex data type in programming. They're everywhere, so their use in SOAP is critical. We covered the details of SOAP arrays back in Chapter 3, so you should be aware of how arrays are encoded. So let's get right into writing some Java code for services that use arrays.

We've been working with stock market examples, so let's stick with that theme. It might be useful to have a service that returns information about a collection of stock symbols. It might provide the total volume of shares traded for the day, the average trading price of those stocks, the number of stocks trading higher for the day, etc. There are lots of possibilities. Let's start out with a service that returns the total number of shares traded for the day. The service is called `urn:BasicTradingService`, and it has a method called `getTotalVolume`. Here is the Java class that implements the service:

```
package javasoap.book.ch5;
public class BasicTradingService {

   public BasicTradingService(  ) {}

   public int getTotalVolume(String[] stocks) {

      // get the volumes for each stock from some
      // data feed and return the total
      int total = 345000;
      return total;
   }
}
```

The `BasicTradingService` class contains the method `getTotalVolume( )`, which returns the total number of shares traded. Since we're not going to access a data feed, we'll just return a made-up value. The method returns an integer and takes a single parameter called `stocks` that is an array of `String` values. The strings in the array are the stock symbols; in a real application, we'd retrieve the volume for each stock from our data feed and return the total for all the stocks in the array.

Apache SOAP has built-in support for arrays, so you don't need to do anything special on the service side. This also means that there's nothing new in the deployment descriptor; it's built just like it was in the previous chapter. Here is the deployment descriptor for the `urn:BasicTradingService` service:

```
<isd:service
        xmlns:isd="http://xml.apache.org/xml-soap/deployment"
        id="urn:BasicTradingService">
   <isd:provider type="java"
        scope="Application"
        methods="getTotalVolume">
     <isd:java class="javasoap.book.ch5.BasicTradingService"
        static="false"/>
   </isd:provider>
   <isd:faultListener>org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

You can deploy the service using this deployment descriptor, or you can use the Apache SOAP Admin tool from your browser.

Now let's write a client application that accesses the service. This application is similar to some of the examples from the previous chapter; it differs only in that the parameter we're passing is an array of `String` instances, rather than a single object. Here is the code:

```
package javasoap.book.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class VolumeClient {
    public static void main(String[] args) throws Exception {

        URL url = new URL("http://georgetown:8080/soap/servlet/rpcrouter");

        Call call = new Call(  );
        call.setTargetObjectURI("urn:BasicTradingService");

        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
        String[] stocks = { "MINDSTRM", "MSFT", "SUN" };
        Vector params = new Vector(  );
        params.addElement(new Parameter("stocks",
                            String[].class, stocks, null));
        call.setParams(params);

        try {
            call.setMethodName("getTotalVolume");
            Response resp = call.invoke(url, "");
            Parameter ret = resp.getReturnValue(  );
            Object value = ret.getValue(  );
            System.out.println("Total Volume is " + value);
        }
        catch (SOAPException e) {
            System.err.println("Caught SOAPException (" +
                        e.getFaultCode(  ) + "): " +
                        e.getMessage(  ));
        }
    }
}
```

We passed `String[].class` as the second parameter of the `Parameter` constructor. That identifies the `stocks` variable as an array of strings. That's it. Nothing else is required. If you run this example, the output should be:

```
Total Volume is 345000
```

Let's take a look at the SOAP envelope that was passed to the server for the invocation of the `getTotalVolume` service method:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <SOAP-ENV:Body>
    <ns1:getTotalVolume xmlns:ns1="urn:BasicTradingService"
      SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">
      <stocks
        xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
        xsi:type="ns2:Array"
        ns2:arrayType="xsd:string[3]">
          <item xsi:type="xsd:string">MINDSTRM</item>
          <item xsi:type="xsd:string">MSFT</item>
          <item xsi:type="xsd:string">SUN</item>
      </stocks>
    </ns1:getTotalVolume>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The `stocks` element represents the string array parameter that we passed to the service method. It is typed as an array by setting the `xsi:type` attribute to `ns2:Array`. The `ns2` namespace identifier was defined on the previous line. Next, the `ns2:arrayType` attribute is assigned a value of `xsd:string[3]`. This means that this is an array of size 3, and that each element of the array is an `xsd:string`. There are three child elements of `stocks`, each one named `item`. Remember that the name used for the array elements doesn't matter, and that different SOAP implementations use different schemes for naming these elements. Each element is explicitly typed by setting the `xsi:type` attribute to the value `xsd:string`.

This example uses a homogeneous array, i.e., all of the elements of the array are instances of the same type. You may have occasion to use heterogeneous arrays as well, so let's look at that possibility. In Java, arrays are often used as parameters to methods that, in other languages, would have a variable-length parameter list. For instance, the `printf( )` function in the *C* language doesn't have a fixed number of parameters. Even though Java doesn't support this capability, you can simulate it by passing your parameter values in an array. An array can be of any size, and the array elements aren't required to have the same type.

Let's add a method to the `urn:BasicTradingService` that takes a single heterogeneous array as a parameter. The method is called `executeTrade`. Its parameter is an array containing the stock symbol, the number of shares to trade, and a flag indicating whether it's a buy or sell

order (`true` means buy). The return value is a string that describes the trade.[1] Here is the modified `BasicTradingService` class:

```
package javasoap.book.ch5;
public class BasicTradingService {

   public BasicTradingService(  ) {
   }
   public int getTotalVolume(String[] stocks) {

      // get the volumes for each stock from some
      // data feed and return the total

      int total = 345000;
      return total;
   }
   public String executeTrade(Object[] params) {
      String result;
      try {
         String stock = (String)params[0];
         Integer numShares = (Integer)params[1];
         Boolean buy = (Boolean)params[2];
         String orderType = "Buy";
         if (false == buy.booleanValue(  )) {
            orderType = "Sell";
         }
         result = (orderType + " " + numShares + " of " + stock);
      }
      catch (ClassCastException e) {
         result = "Bad Parameter Type Encountered";
      }
      return result;
   }
}
```

There is only one parameter for the `executeTrade( )` method, an `Object[]` called `params`. The objects in this array must be cast to their corresponding types: a `String`, an `Integer`, and a `Boolean`. I like to put class casts inside a `try/catch` block in case the caller makes a mistake. That way I can do something useful if the method is called incorrectly, even if that means simply returning a description of the error. In Chapter 7, we'll look at generating SOAP faults for situations like this. The information passed in the array is used to generate a string that describes the parameters, and that string is stored in the `result` variable that is returned to the caller.

Now we can modify the client application so that it passes an appropriate `Object[]` as the parameter to the `executeTrade` service method. The `multiParams` variable is declared as an `Object[]`, and is populated with the `String` MINDSTRM, an `Integer` with the value of 100, and a `Boolean` with the value of `true`. Since we're using an array of Java `Object` instances, we don't use Java primitives as elements of the array. Instead we wrap those primitive values in their Java object equivalents. The second parameter of the `Parameter` constructor is `Object[].class`, which is the class for an array of object instances.

---

[1] There are certainly other ways to design the interface to a method like this, and this is not the design I'd choose. This situation probably calls for using a custom class or Java bean. However, the approach I've used in this example demonstrates the use of heterogeneous arrays.

```
package javasoap.book.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class TradingClient {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call(  );
      call.setTargetObjectURI("urn:BasicTradingService");

      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      Object[] multiParams = { "MINDSTRM", new Integer(100),
                                   new Boolean(true) };
      Vector params = new Vector(  );
      params.addElement(new Parameter("params",
                       Object[].class, multiParams, null));
      call.setParams(params);

      try {
         call.setMethodName("executeTrade");
         Response resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue(  );
         Object value = ret.getValue(  );
         System.out.println("Trade Description: " + value);
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                     e.getFaultCode(  ) + "): " +
                     e.getMessage(  ));
      }
   }
}
```

If all goes well, the result of executing the `executeTrade( )` service method is:

```
Trade Description: Buy 100 of MINDSTRM
```

We could force the service object down another path by changing the order of the parameters in the `multiParams` array. In this case, we would encounter a class cast exception, and the method would return an error string.

Here is the SOAP envelope for the proper invocation of the `executeTrade( )` service method:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <SOAP-ENV:Body>
    <ns1:executeTrade xmlns:ns1="urn:BasicTradingService"
      SOAP-ENV:encodingStyle=
         "http://schemas.xmlsoap.org/soap/encoding/">
```

```
      <params
          xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
          xsi:type="ns2:Array" ns2:arrayType="xsd:anyType[3]">
        <item xsi:type="xsd:string">MINDSTRM</item>
        <item xsi:type="xsd:int">100</item>
        <item xsi:type="xsd:boolean">true</item>
      </params>
    </ns1:executeTrade>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The `params` element is typed by assigning the `xsi:type` attribute the value of `ns2:Array`. The only difference from the homogeneous case is that every array element has a different value assigned to the `ns2:arrayType` attribute. The value `xsd:anyType[3]` indicates that the array contains 3 elements, each of which can be of any valid data type.[2]

Now let's take a look at passing arrays as parameters using GLUE. The `BasicTradingService` class can be deployed in GLUE without modification. We'll use a simple Java application to get this service started:

```
package javasoap.book.ch5;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class BasicTradingApp {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context(  );
      context.addProperty("activation", "application");
      context.addProperty("namespace",
                               "urn:BasicTradingService");
      Registry.publish("urn:BasicTradingService",
         javasoap.book.ch5.BasicTradingService.class, context );
   }
}
```

Compile and execute the application, and the service is deployed. Now let's write a simple example to access the service using the GLUE API. First let's look at the interface to the service, `IBasicTradingService`:

```
package javasoap.book.ch5;
public interface IBasicTradingService {
  int getTotalVolume(String[] symbols);
  String executeTrade(Object[] params);
}
```

Now we can write an application that binds to the service and calls both its methods:

---

[2] In Chapter 3, we talked about using `ur-type` to represent any possible data type. Here we see the use of `anyType` for that purpose. The XML Schema Part 0 recommendation, dated May 2, 2001, explains this in section 2.5.4 as follows: "The anyType represents an abstraction called the `ur-type` which is the base type from which all simple and complex types are derived. An anyType type does not constrain its content in any way."

```
package javasoap.book.ch5;
import electric.registry.RegistryException;
import electric.registry.Registry;
public class BasicTradingClient {
   public static void main(String[] args) throws Exception
   {
      try {
        IBasicTradingService srv =
          (IBasicTradingService)Registry.bind(
            "http://georgetown:8004/glue/urn:BasicTradingService.wsdl",
            IBasicTradingService.class);
        String[] stocks = { "MINDSTRM", "MSFT", "SUN" };
        int total = srv.getTotalVolume(stocks);
        System.out.println("Total Volume is " + total);
        Object[] multiParams = { "MINDSTRM", new Integer(100),
                                  new Boolean(true) };
        String desc = srv.executeTrade(multiParams);
        System.out.println("Trade Description: " + desc);
      }
      catch (RegistryException e)
      {
         System.out.println(e);
      }
   }
}
```

As we've seen before, GLUE allows us to use familiar Java programming syntax without having to think about the underlying SOAP constructs. This holds true for the passing of array parameters as well. Everything is pretty much handled for us after the interface is bound to the service.

There are some interesting things to see in the SOAP request envelopes generated by this example. Here is the SOAP envelope for the `getTotalVolume( )` method invocation:

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getTotalVolume xmlns:n='urn:BasicTradingService'>
        <arg0 href='#id0'/>
      </n:getTotalVolume>
      <id0 id='id0' soapenc:root='0'
        xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
        xsi:type='soapenc:Array' soapenc:arrayType='xsd:string[3]'>
       <i xsi:type='xsd:string'>MINDSTRM</i>
       <i xsi:type='xsd:string'>MSFT</i>
       <i xsi:type='xsd:string'>SUN</i>
      </id0>
    </soap:Body>
</soap:Envelope>
```

The `Body` element starts off like we've seen before; the envelope is qualified by a namespace identifier, `soap`, which represents the namespace of the SOAP envelope. The first child element of the SOAP body is `getTotalVolume`, which of course is the name of the service method being invoked. `getTotalVolume` is namespace-qualified using the name of the

service. The only child element of `getTotalVolume` is `arg0`, which represents the parameter passed to the method. But this isn't the array we passed; it's a reference to the array. This is a significant difference between the ways that GLUE and the Apache SOAP API generate this call. Apache SOAP puts the array in the envelope as a child element of `getTotalVolume`, and GLUE uses a reference and serializes the array after the `getTotalVolume` element terminates. So the parameter is serialized as `arg0`, and includes an `href` attribute with the value `#id0`. No data is included, as the array resides elsewhere.

The array that we passed as a parameter follows the `getTotalVolume` element. It's named `id0`, although the element name itself, which is generated by GLUE, is not important. The `id` attribute is assigned a value of `id0`, which coincides with the `href` value used in the `getTotalVolume` element. GLUE generates the `soapenc:root` attribute with a value of 0, meaning that this element is not considered the root of an object graph. (GLUE seems to include that attribute automatically.) Next we see a declaration of a namespace identifier called `ns2` that seems to identify the internal GLUE package for `java.lang`; however, the `ns2` namespace identifier is never used. The `xsi:type` and `soapenc:arrayType` attributes are set up in the same way as in the Apache SOAP examples. Finally, the elements of the array are serialized. The only difference between this example and the one generated by Apache SOAP is in the name of the array elements themselves. Apache SOAP named these elements `item`, and GLUE named them `i`. The names don't matter; the result is the same.

This example gives us a good opportunity to see two equally valid ways to serialize arrays. It's important that SOAP implementations understand these different styles if they are to interoperate. This is one of the reasons we keep showing the SOAP envelopes generated by the examples. Becoming familiar with the various styles of serialization will help you down the road if you run into problems getting applications based on different implementations to communicate correctly. (Did I say *if*?)

Let's take a look at the SOAP envelope for the call to the `executeTrade` service method. This method takes a heterogeneous array as a parameter. It too uses a reference to a separately serialized array, this time encoded as `xsd:anyType`:

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:executeTrade xmlns:n='urn:BasicTradingService'>
        <arg0 href='#id0'/>
      </n:executeTrade>
      <id0 id='id0' soapenc:root='0'
        xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
        xsi:type='soapenc:Array' soapenc:arrayType='xsd:anyType[3]'>
        <i xsi:type='xsd:string'>MINDSTRM</i>
        <i xsi:type='xsd:int'>100</i>
        <i xsi:type='xsd:boolean'>true</i>
      </id0>
    </soap:Body>
</soap:Envelope>
```

5.2 Returning Arrays

So far we've been passing arrays as parameters. Now let's use an array as the return value of a service method. We'll add a method to our service called `getMostActive( )`, which returns a `String[]` that contains the symbols for the most actively traded stocks of the day. Here's the new version of the `BasicTradingService` class:

```java
package javasoap.book.ch5;
public class BasicTradingService {

   public BasicTradingService( ) {
   }
   public String[] getMostActive( ) {

      // get the most actively traded stocks
      String[] actives = { "ABC", "DEF", "GHI", "JKL" };
      return actives;
   }
   public int getTotalVolume(String[] stocks) {

      // get the volumes for each stock from some
      // data feed and return the total
      int total = 345000;
      return total;
   }
   public String executeTrade(Object[] params) {
      String result;
      try {
         String stock = (String)params[0];
         Integer numShares = (Integer)params[1];
         Boolean buy = (Boolean)params[2];
         String orderType = "Buy";
         if (false == buy.booleanValue( )) {
            orderType = "Sell";
         }
         result = (orderType + " " + numShares + " of " + stock);
      }
      catch (ClassCastException e) {
         result = "Bad Parameter Type Encountered";
      }
      return result;
   }
}
```

Since we're not really calling a data feed, we just stuff a few phony stock symbols into an array and return it. Go ahead and redeploy the service now. Calling this service method from an Apache SOAP client is simple. There are no parameters to the service method, so we just have to set up the call and invoke it:

```
package javasoap.book.ch5;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class MostActiveClient
{
  public static void main(String[] args) throws Exception
  {
    URL url = new
      URL("http://georgetown:8080/soap/servlet/rpcrouter");
    Call call = new Call(  );
    call.setTargetObjectURI("urn:BasicTradingService");
    call.setMethodName("getMostActive");
    Response resp;
    try {
      resp = call.invoke(url, "");
      Parameter ret = resp.getReturnValue(  );
      String[] value = (String[])ret.getValue(  );
      int cnt = value.length;
      for (int i = 0; i < cnt; i++) {
          System.out.println(value[i]);
      }
    }
    catch (SOAPException e) {
      System.err.println("Caught SOAPException (" +
                          e.getFaultCode(  ) + "): " +
                          e.getMessage(  ));
    }
  }
}
```

We cast the return value of `ret.getValue` to a `String[]`, since that's the return type we're expecting. In past examples we were able to leave the value as an `Object` instance because we relied on the object's `toString( )` method to display the value. In this case we need to iterate over the array, so it's necessary to cast the value to the appropriate array type. After that, we just find the array length and then loop over the array values, printing each one as we get to it. If you run this example you should see the following output:

```
ABC
DEF
GHI
JKL
```

The SOAP envelope returned by this method invocation is pretty straightforward:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:getMostActiveResponse
        xmlns:ns1="urn:BasicTradingService"
        SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">
```

```
        <return
          xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
          xsi:type="ns2:Array" ns2:arrayType="xsd:string[4]">
            <item xsi:type="xsd:string">ABC</item>
            <item xsi:type="xsd:string">DEF</item>
            <item xsi:type="xsd:string">GHI</item>
            <item xsi:type="xsd:string">JKL</item>
        </return>
      </ns1:getMostActiveResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

To deploy this version of the `BasicTradingService` class in GLUE, we can use our old `BasicTradingApp` class. We can modify the Java interface, `IBasicTradingService`, to include the new method:

```
package javasoap.book.ch5;
public interface IBasicTradingService {
  int getTotalVolume(String[] symbols);
  String executeTrade(Object[] params);
  String[] getMostActive(  );
}
```

Now we modify the application `BasicTradingClient` to include a call to the `getMostActive( )` method, and then iterate over the values in the array and print them out. When using GLUE we don't have to cast the return value to a `String[]` because, unlike the Apache SOAP example, the `getMostActive( )` method of the interface is defined to return the proper type. Here's the modified code:

```
package javasoap.book.ch5;
import electric.registry.RegistryException;
import electric.registry.Registry;
public class BasicTradingClient {
   public static void main(String[] args) throws Exception
   {
      try {
        IBasicTradingService srv = (IBasicTradingService)Registry.bind(
          "http://georgetown:8004/glue/urn:BasicTradingService.wsdl",
          IBasicTradingService.class);
        String[] stocks = { "MINDSTRM", "MSFT", "SUN" };
        int total = srv.getTotalVolume(stocks);
        System.out.println("Total Volume is " + total);
        Object[] multiParams = { "MINDSTRM", new Integer(100),
                                    new Boolean(true) };
        String desc = srv.executeTrade(multiParams);
        System.out.println("Trade Description: " + desc);
        String[] actives = srv.getMostActive(  );
        int cnt = actives.length;
        for (int i = 0; i < cnt; i++) {
           System.out.println(actives[i]);
        }
      }
      catch (RegistryException e)
      {
        System.out.println(e);
      }
   }
}
```

If you run this example, you'll get the following output:

```
Total Volume is 345000
Trade Description: Buy 100 of MINDSTRM
ABC
DEF
GHI
JKL
```

GLUE uses the same serialization technique for arrays as return values that we saw earlier for array parameters; it uses a reference to a separately serialized array as the actual return value, and it references the actual array data. The SOAP envelope returned when invoking the `getMostActive( )` method is:

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getMostActiveResponse
        xmlns:n='urn:BasicTradingService'>
        <Result href='#id0'/>
      </n:getMostActiveResponse>
      <id0 id='id0' soapenc:root='0'
        xmlns:ns2='http://www.themindelectric.com/package/java.lang/'
        xsi:type='soapenc:Array' soapenc:arrayType='xsd:string[4]'>
        <i xsi:type='xsd:string'>ABC</i>
        <i xsi:type='xsd:string'>DEF</i>
        <i xsi:type='xsd:string'>GHI</i>
        <i xsi:type='xsd:string'>JKL</i>
      </id0>
    </soap:Body>
</soap:Envelope>
```

5.3 Passing Custom Types as Parameters

As Java programmers, we certainly don't restrict ourselves to the classes found in the standard Java packages. A substantial part of our code exists as custom types, Java classes of our own creation that embody the functionality and characteristics of the systems we're building.

Consider the design of a Java class that contains all of the data necessary to specify a stock trade. This new class might contain the symbol for the stock being traded, the number of shares to trade, and an indication of the order type (buy or sell). When designing such a class, it's important to view it in the context of the larger system. That kind of analysis yields clues that can lead to decisions regarding the properties and behaviors to be given to the class. We all do this kind of work all the time; it's called software design. The result is a Java class that contains methods for accessing properties and behavior. Since SOAP is a data transport, we're interested in the properties of the class. That's what we want to transmit over the wire.

One common way to express the properties of a Java class is to use the JavaBeans design patterns. These patterns specify a naming convention to be used for the class's access methods. You may not be familiar with JavaBeans, but I bet you've seen this pattern many

times. Here's how the property accessor pattern is described in the O'Reilly book *Developing Java Beans*:

> The methods used for getting and setting property values should conform to the standard design pattern for properties. These methods are allowed to throw checked exceptions if desired, but this is optional. The method signatures are as follows:
>
> ```
> public void set<PropertyName>(<PropertyType> value);
> public <PropertyType> get<PropertyName>(  );
> ```
>
> The existence of a matching pair of methods that conform to this pattern represents a read/write property with the name <PropertyName> of the type <PropertyType>. If only the get method exists, the property is considered to be read only, and if only the set method exists the property is considered to be write only. In the case where the <PropertyType> is boolean, the get method can be replaced or augmented with a method that uses the following signature:

```
public boolean is<PropertyName>(  );
```

If you follow this pattern for naming property accessors, the accessor methods can be determined at runtime by using the Java reflection mechanism.[3] This is a convenient way for SOAP implementations to access the data values of a Java class instance in order to serialize the data in a SOAP message. It turns out that both Apache SOAP and GLUE take advantage of reflection. This means that all you need to do is follow a well-established naming convention, and you'll be well on your way to using custom classes in SOAP. Of course, there's a little more to it than that, so let's dig in.

First let's define a stock trade in terms of data that we want it to contain. It needs to have a stock symbol to represent the stock being traded; the symbol should be a string type element. It needs a `boolean` indicator that specifies whether the order is buy or sell. Lastly, it needs an integer that contains the number of shares to be traded. (In a real-world application, it might also contain various security credentials, the names of the purchaser and the broker, and many other things. Alternatively, this ancillary data could be represented in other objects.) Here's what an XML schema snippet for the stock trade might look like:

```
<element name="StockTrade" type="StockTrade"/>
<complexType name="StockTrade">
    <element name="symbol" type="xsd:string"/>
    <element name="buy" type="xsd:boolean"/>
    <element name="numshares" type="xsd:int"/>
</complexType>
```

Let's create a custom Java class for specifying a stock trade called `StockTrade_ClientSide`. Normally I'd name this class `StockTrade`, but I want to make it clear that I'll be using this class on the client side of the example. We'll be writing a similar class to be used on the server side that will have a corresponding name. I'm doing this to point out that you are not required to use the same Java class on both sides of the message transaction. In fact, it probably won't make sense to use the same class, and it often won't even be possible.

---

[3] JavaBeans provides for other ways to accomplish this, but those are not within the scope of this book.

`StockTrade_ClientSide` has three read/write properties named `Symbol`, `Buy`, and `NumShares` that represent the stock symbol, the buy/sell indicator, and the number of shares to trade, respectively.

```
package javasoap.book.ch5;
public class StockTrade_ClientSide {
   String   _symbol;
   boolean  _buy;
   int      _numShares;
   public StockTrade_ClientSide(  ) {
   }
   public StockTrade_ClientSide(String symbol,
                      boolean buy, int numShares) {
      _symbol = symbol;
      _buy = buy;
      _numShares = numShares;
   }
   public String getSymbol(  ) {
      return _symbol;
   }
   public void setSymbol(String symbol) {
      _symbol = symbol;
   }
   public boolean isBuy(  ) {
      return _buy;
   }
   public void setBuy(boolean buy) {
      _buy = buy;
   }
   public int getNumShares(  ) {
      return _numShares;
   }
   public void setNumShares(int numShares) {
      _numShares = numShares;
   }
}
```

Now let's create a `StockTrade_ServerSide` class to represent the stock trade on the server side. Just to be sure that this class is different from its client-side counterpart, let's eliminate the constructor that takes parameters. And for good measure, let's also change the names of the class variables and the order in which they appear.

```
package javasoap.book.ch5;
public class StockTrade_ServerSide {
   int      _shares;
   boolean  _buyOrder;
   String   _stock;
   public StockTrade_ServerSide(  ) {
   }
   public String getSymbol(  ) {
      return _stock;
   }
   public void setSymbol(String stock) {
      _stock = stock;
   }
   public boolean isBuy(  ) {
      return _buyOrder;
   }
```

```
   public void setBuy(boolean buyOrder) {
      _buyOrder = buyOrder;
   }
   public int getNumShares(  ) {
      return _shares;
   }
   public void setNumShares(int shares) {
      _shares = shares;
   }
}
```

We can add a new method to the `urn:BasicTradingService` service called `executeStockTrade( )`, which takes a stock trade as a parameter. The return value from this method is a string that describes the order. Here's the modified `BasicTradingService` class. We can take advantage of the `executeTrade( )` method that already exists in this class. In the new method, `executeStockTrade( )`, we build an `Object` array from the three properties of the `trade` parameter, and we pass that array to the `executeTrade( )` method.

```
package javasoap.book.ch5;
public class BasicTradingService {

   public BasicTradingService(  ) {
   }
   public String executeStockTrade(StockTrade_ServerSide trade) {
      Object[] params = new Object[3];
      params[0] = trade.getSymbol(  );
      params[1] = new Integer(trade.getNumShares(  ));
      params[2] = new Boolean(trade.isBuy(  ));
      return executeTrade(params);
   }
   public String[] getMostActive(  ) {

      // get the most actively traded stocks
      String[] actives = { "ABC", "DEF", "GHI", "JKL" };
      return actives;
   }
   public int getTotalVolume(String[] stocks) {

      // get the volumes for each stock from some
      // data feed and return the total

      int total = 345000;
      return total;
   }
   public String executeTrade(Object[] params) {
      String result;
      try {
         String stock = (String)params[0];
         Integer numShares = (Integer)params[1];
         Boolean buy = (Boolean)params[2];
         String orderType = "Buy";
         if (false == buy.booleanValue(  )) {
            orderType = "Sell";
         }
         result = (orderType + " " + numShares + " of " + stock);
      }
```

```
        catch (ClassCastException e) {
            result = "Bad Parameter Type Encountered";
        }
        return result;
    }
}
```

To deploy the service, we need to map the custom type to the Java class that implements that type. We need to give the custom type a name and qualify it with an appropriate namespace. This is not unlike the process we'd use to declare a service. This mapping takes place in the deployment descriptor, within the `isd:mappings` section. Here's the deployment descriptor we'll use to deploy the service in Apache SOAP:

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:BasicTradingService">
  <isd:provider
     type="java"
     scope="Application"
     methods="getTotalVolume getMostActive executeTrade executeStockTrade">
    <isd:java
       class="javasoap.book.ch5.BasicTradingService"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
  <isd:mappings>
    <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:BasicTradingService" qname="x:StockTrade"
       javaType="javasoap.book.ch5.StockTrade_ServerSide"
       java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
      xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

This is important, so let's summarize the mapping before we go any further. The mapping of a custom type to a Java class is associated with an encoding style as well as a fully qualified type name. A Java class that implements the custom type is specified, as well as the utility classes used to perform serialization and deserialization. Now let's look at the details of the example.

Most of the deployment descriptor should look familiar; it's similar to the ones shown in Chapter 4. The difference is that we now have an entry in the `isd:mappings` section. There is one entry, `isd:map`, that describes the mapping of the stock trade type to the `StockTrade_ServerSide` class. The `isd:map` element has no data; all the information is supplied as attributes. The first attribute, `encodingStyle`, specifies the encoding style associated with the serialization of the type. Next, a namespace identifier, `x`, is assigned the value `urn:BasicTradingService`. I'm using the name of the service as the namespace qualifier for the custom type; however, you don't have to do this, and in fact may not want to in many instances. For example, if you have custom types that are used by multiple services, or if you're using custom types whose definitions are specified by a third party, then you'd certainly want to qualify the custom type using the appropriate namespace. The next attribute, `qname`, specifies the fully qualified name of the type being mapped. The value assigned is

`x:StockTrade`, which is the same `StockTrade` namespace qualified using the service name. The `javaType` attribute specifies the server-local Java class used to implement the type; on the server side we're using `javasoap.book.ch5.StockTrade_ServerSide`. The last two attributes, `java2XMLClassName` and `xml2JavaClassName`, tell Apache SOAP which local Java classes to use to perform the serialization and deserialization, respectively. Apache SOAP comes with a custom serializer/deserializer class that can convert between custom XML types and Java classes that conform to the JavaBeans property accessor pattern. It can handle both serialization and deserialization, which is why we use it for both attributes. In the next chapter we'll look at creating custom type serializers.

Now that we have a deployment descriptor, we can go ahead and redeploy the `urn:BasicTradingService`. Once we do that, our service is ready to accept `executeStockTrade` method invocations. However, we need to do some setup work in the client application as well. Let's take a look at the client application:

```
package javasoap.book.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
public class StockTradeClient
{
  public static void main(String[] args) throws Exception
  {
    URL url = new
       URL("http://georgetown:8080/soap/servlet/rpcrouter");
    Call call = new Call(  );
    SOAPMappingRegistry smr = new SOAPMappingRegistry(  );
    call.setSOAPMappingRegistry(smr);
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setTargetObjectURI("urn:BasicTradingService");
    call.setMethodName("executeStockTrade");
    BeanSerializer beanSer = new BeanSerializer(  );
    // Map the Stock Trade type
    smr.mapTypes(Constants.NS_URI_SOAP_ENC,
       new QName("urn:BasicTradingService", "StockTrade"),
       StockTrade_ClientSide.class, beanSer, beanSer);
    // create an instance of the stock trade
    StockTrade_ClientSide trade =
         new StockTrade_ClientSide("XYZ", false, 350);
    Vector params = new Vector(  );
    params.addElement(new Parameter("trade",
                         StockTrade_ClientSide.class, trade, null));
    call.setParams(params);
    Response resp;
    try {
      resp = call.invoke(url, "");
      Parameter ret = resp.getReturnValue(  );
      Object desc = ret.getValue(  );
      System.out.println("Trade Description: " + desc);
    }
```

```
      catch (SOAPException e) {
        System.err.println("Caught SOAPException (" +
                          e.getFaultCode(  ) + "): " +
                          e.getMessage(  ));
      }
   }
}
```

After the `Call` object is created, we create an instance of `org.apache.soap.encoding.SOAPMappingRegistry` called `smr`. This class holds the mappings of custom types to Java classes, and gets passed to the `Call` object using the `setSOAPMappingRegistry(  )` method of our `Call` object. We haven't had to do this in previous examples because the `Call` object creates a mapping registry for itself if one isn't passed to it. The `SOAPMappingRegistry(  )` contains all of the predefined mappings, such as those we took advantage of for arrays. We'll add our mapping to it shortly.

Next we call the `call.setEncodingStyleURI(  )` method. This method specifies the overall encoding style to be used for the custom type parameters. The constant `NS_URI_SOAP_ENC`, from the `org.apache.soap.Constants` class, represents the `http://schemas.xmlsoap.org/soap/encoding/` namespace that we've been using thus far. This should be the same encoding style namespace that we specified in the deployment descriptor for the `StockTrade` type. The `setTargetObjectURI(  )` and `setMethodName(  )` methods are used in the same way as in previous examples. The next step is to create an instance of `org.apache.soap.encoding.soapenc.BeanSerializer`; we can use this standard serializer because our custom type conforms to the JavaBeans property accessor pattern. Now we can establish the mapping by calling `smr.mapTypes(  )`. The first parameter is `Constants.NS_URI_SOAP_ENC`, which specifies that the encoding style used for this mapping is the standard SOAP encoding. You may be wondering why we need to do this if we've already specified the encoding style for the entire call earlier. The reason is simple: this parameter gives you the opportunity to override the encoding style used for this particular mapping. However, if you use `null` as the parameter value, you'll end up with a mapping that attempts to use the `null` namespace for the encoding style, which is not correct. When we look at the SOAP envelope for this message, you'll see that since the encoding style is the same as that of the overall call, the `encodingStyle` attribute will not be repeated for this serialized parameter. If the encoding style were not the same as that of the `Call`, it would appear as an attribute of the parameter.

The next parameter of `smr.mapTypes(  )` is an instance of `org.apache.soap.util.xml.Qname`, which represents a fully qualified name (one that includes a namespace qualifier followed by a name). We use `urn:BasicTradingService` as the namespace and `StockTrade` as the name. The third parameter is `StockTrade_ClientSide.class`, the Java class that implements the custom type. The next two parameters are instances of the serializer and deserializer that will be used for this type. We use the `beanSer` object that we created earlier for both, as the `org.apache.soap.encoding.soapenc.BeanSerializer` class implements both serialization and deserialization.

The rest is pretty simple. We create an instance of `StockTrade_ClientSide`, taking advantage of the parameterized constructor to set its property values. Then we set up a `Vector` of parameters, just as we've done in earlier examples. If you run this example, you should see the following output:

```
Trade Description: Sell 350 of XYZ
```

Let's take a look at the SOAP envelope that was transmitted. The relevant part of the envelope begins with the `trade` element, representing the custom type parameter passed to the `executeStockTrade` service method. The value assigned to `xsi:type` is `ns1:StockTrade`. The `ns1` namespace identifier is declared to be `urn:BasicTradingService` in the parent `executeStockTrade` element. And `StockTrade` is the name we specified for our custom type. There are three child elements of the `trade` element, each one corresponding to the properties of the `StockTrade` custom type. The name of each element corresponds exactly to the property name. This is crucial, as Apache SOAP is going to use Java reflection to find the set methods of the associated Java class. Therefore, the names in the envelope must match the names used by the class. Each one of these property elements is explicitly typed, and those types have to coincide with the types of the corresponding properties as well.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:executeStockTrade xmlns:ns1="urn:BasicTradingService"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <trade xsi:type="ns1:StockTrade">
        <numShares xsi:type="xsd:int">350</numShares>
        <buy xsi:type="xsd:boolean">false</buy>
        <symbol xsi:type="xsd:string">XYZ</symbol>
      </trade>
    </ns1:executeStockTrade>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Now let's see how custom types are handled in GLUE, using the `BasicTradingService` class without modification. We need to add the `executeStockTrade( )` method to the `IBasicTradingService` interface. We can deploy the service as we did before. Here's the new version of `IBasicTradingService`:

```
package javasoap.book.ch5;
public interface IBasicTradingService {
  int getTotalVolume(String[] symbols);
  String executeTrade(Object[] params);
  String[] getMostActive(  );
  String executeStockTrade(StockTrade_ClientSide trade);
}
```

Writing applications that access services has been easy using the GLUE API because we haven't dealt directly with the SOAP constructs. We should be able to follow that same pattern here, but there's a problem. The `executeStockTrade( )` method of the `IBasicTradingInterface` interface says that the parameter is an instance of `StockTrade_ClientSide`. But the `executeStockTrade` method of the `BasicTradingService` class that implements the service uses a parameter of `StockTrade_ServerSide`. So there's a mismatch, albeit by design. By default, GLUE looks for the same class on the client and server sides. If we had used the `StockTrade_ServerSide` class in our client application instead of the `StockTrade_ClientSide` class, all would work perfectly. We're not going to do that, so just take my word for it that it works (or better yet, try it for yourself). We'll have to take another approach.

Whenever I develop distributed systems, I'm happy to share Java interfaces between both client and server code bases. However, I don't like to share Java classes, especially if those classes contain code that is specific to either the client or the server. In this example, we could rework the design of our stock trade classes to come up with something that contains the relevant data without any of the functionality of either the client or the server. We would want both the client and the server to use the class from the same package, so as the implementer of the basic trading web service, we'd have to distribute the package containing such a class to the developers' client applications. That would work, and it's not uncommon in practice. Getting back to the notion of sharing Java interfaces instead of classes, we could create an interface for the trade data, and have both the server and client classes implement that interface. That's a nice clean way to share between server and client code bases without sharing any actual executable code. Again, this interface would reside in a package available to both server and client systems. This mechanism is expected to be supported in a future version of GLUE (which might already be available by the time you read this).

GLUE does support another mechanism for making this stuff work. This mechanism doesn't require you to modify the server side, and the work involved on the client side is trivial. We're going to create a new package for this client example, because our work will create some files that have the same names as those we created earlier. The new package prevents us from overwriting files when running both client and server on the same machine. So be aware that this example is part of a new package called `javasoap.book.ch5.client`, and the files we'll be developing need to be in the corresponding directory for that package.

I've purposely avoided discussion of WSDL so far, even though GLUE is based entirely on WSDL. However, the first step in this process makes use of GLUE's `wsdl2java` utility, which generates Java code based on a WSDL file. So where does the WSDL come from? The GLUE server generates it automatically. The client-side applications based on the GLUE API have been taking advantage of this all along. `wsdl2java` generates the Java interface for binding to the service, a helper class, a Java data structure class that represents the data fields of the custom type we're working with, and a map file. The map file is essentially a schema definition for the custom type; it tells GLUE how to map the fields of the Java data structure class to the fields of the custom type. GLUE uses a number of mechanisms for handling custom type mapping; in this case, the map file defines the mapping explicitly, rather than basing the mapping on Java reflection.

So let's go ahead and generate the files for the client application. Enter the following command, making sure you are in the directory for the `javasoap.book.ch5.client` package:

```
wsdl2java http://georgetown:8004/glue/urn:BasicTradingService.wsdl
    -p javasoap.book.ch5.client
```

The parameter to the `wsdl2java` utility is the full URL of the service WSDL, just like we've been using in the `bind( )` methods in previous examples. The `-p` option tells the utility the name of the local package for which code should be generated: `javasoap.book.ch5.client`. The output from `wsdl2java` consists of four files. The first one is *IBasicTradingService.java*, which contains the Java interface for binding to the service. We've been writing this one by hand up until now; let's take a look at the one generated by `wsdl2java`:

```
// generated by GLUE
package javasoap.book.ch5.client;
public interface IBasicTradingService
   {
   String executeStockTrade( StockTrade_ServerSide arg0 );
   String[] getMostActive(  );
   int getTotalVolume( String[] arg0 );
   String executeTrade( Object[] arg0 );
   }
```

The only differences between this interface definition and the one we wrote ourselves are a different package declaration, the naming of the method parameters, and the `executeStockTrade` taking a parameter of `StockTrade_ServerSide` instead of `StockTrade_ClientSide`. Remember that the earlier version of `IBasicTradingService` won't work for us because GLUE defaults to using the same class on both client and server. At first glance, the use of the `StockTrade_ServerSide` class here seems to violate our desire to completely decouple the server code from the client code. But remember that the package declaration is `javasoap.book.ch5.client`, so this Java interface is not referencing the same `StockTrade_ServerSide` class being used by our server, which belongs to a different package. Let's take a look at the `StockTrade_ServerSide` class generated by `wsdl2java` and placed in the file named *StockTrade_ServerSide.java*:

```
// generated by GLUE
package javasoap.book.ch5.client;
public class StockTrade_ServerSide
   {
   public int _shares;
   public boolean _buyOrder;
   public String _stock;
   }
```

This is only a shadow of the corresponding class in the `javasoap.book.ch5` package that the server side uses. It does, however, properly reflect the data values. We'll be creating an instance of this class in our client application to pass to the service, as this is the type expected by the `executeStockTrade( )` method of the `IBasicTradingService` interface that was generated.

The third file generated by `wsdl2java` is *BasicTradingService.map*, which contains the mapping schema mentioned earlier. You can take a look at the contents of the mapping file on your own; suffice it to say that it contains XML entries that define field mappings between the client-side and server-side Java. The last generated file is *BasicTradingServiceHelper.java*, which contains a helper class for performing the `bind( )` operation. I don't use that helper class, so we'll ignore it here.

It's taken far longer to describe the process than it takes to perform it. Now let's move on to writing the client application:

```
package javasoap.book.ch5.client;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.xml.io.Mappings;
public class StockTradeClient2 {
   public static void main(String[] args) throws Exception
   {
      try {
        Mappings.readMappings("BasicTradingService.map");
        IBasicTradingService srv =
          (IBasicTradingService)Registry.bind(
          "http://georgetown:8004/glue/urn:BasicTradingService.wsdl",
          IBasicTradingService.class);
        StockTrade_ServerSide trade =
                   new StockTrade_ServerSide(  );
        trade._stock = "MINDSTRM";
        trade._buyOrder = true;
        trade._shares = 500;
        String desc = srv.executeStockTrade(trade);
        System.out.println("Trade Description is: " + desc);
      }
      catch (RegistryException e)
      {
         System.out.println(e);
      }
   }
}
```

The critical step is to read the mappings before you perform the bind operation. This is done by calling the `readMappings( )` method of the `electric.xml.io.Mappings` class provided as part of GLUE. After the `bind( )` call, we simply create an instance of `StockTrade_ServerSide`, set its field values, and call the `executeStockTrade` method. The output from running this application should be:

```
Trade Description is: Buy 500 of MINDSTRM
```

Now let's look at the SOAP envelope generated by this application. As we've come to expect, the `executeStockTrade` element is namespace-qualified using the name of the service. The child element `arg0` is a reference to a separately serialized element named `id0`, just as GLUE generated for an array parameter. This example shows us why GLUE has been generating the `ns2` namespace identifier that seems to reference the package where the implementing server-side class resides. Here that namespace is used to qualify the value of the `xsi:type` attribute, with a value corresponding to the name of the implementing class `StockTrade_ServerSide`. The child elements of the `id0` element contain the data fields for the custom type.

```
<soap:Envelope
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
  xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
  soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
```

```
  <soap:Body>
    <n:executeStockTrade xmlns:n='urn:BasicTradingService'>
      <arg0 href='#id0'/>
    </n:executeStockTrade>
    <id0 id='id0' soapenc:root='0'
    xmlns:ns2='http://www.themindelectric.com/package/javasoap.book.ch5/'
     xsi:type='ns2:StockTrade_ServerSide'>
      <_shares xsi:type='xsd:int'>500</_shares>
      <_buyOrder xsi:type='xsd:boolean'>true</_buyOrder>
      <_stock xsi:type='xsd:string'>MINDSTRM</_stock>
    </id0>
  </soap:Body>
</soap:Envelope>
```

5.4 Returning Custom Types

It's equally useful (and equally common) to return custom types from service method calls. We can enhance our trading service by offering a method that takes a single stock symbol as a parameter and returns its high and low trading prices for the day. The classes HighLow_ServerSide and HighLow_ClientSide represent the high/low prices on the server and client, respectively.

```
package javasoap.book.ch5;
public class HighLow_ServerSide {
   public float _high;
   public float _low;
   public HighLow_ServerSide(  ) {
   }
   public HighLow_ServerSide (float high, float low) {
      setHigh(high);
      setLow(low);
   }
   public float getHigh(  ) {
      return _high;
   }
   public void setHigh(float high) {
      _high = high;
   }
   public float getLow(  ) {
      return _low;
   }
   public void setLow(float low) {
      _low = low;
   }
}

package javasoap.book.ch5;
public class HighLow_ClientSide {
   public float _high;
   public float _low;
   public String toString(  ) {
      return "High: " + _high +
        " Low: " + _low;
   }
   public HighLow_ClientSide(  ) {
   }
   public float getHigh(  ) {
      return _high;
   }
```

```
   public void setHigh(float high) {
      _high = high;
   }
   public float getLow(  ) {
      return _low;
   }
   public void setLow(float low) {
      _low = low;
   }
}
```

The server-side class includes a parameterized constructor as a convenience for creating the return value; the client-side class includes a `toString(  )` method to make it easy for our client application to display the contents of the object after it's returned from the server. Let's add a new method to the `BasicTradingService` class called `getHighLow(  )`. That method takes a single string parameter for the stock symbol and returns an instance of `HighLow_ServerSide`. Here's the class with its new method, with the unchanged code omitted:

```
package javasoap.book.ch5;
public class BasicTradingService {
   public BasicTradingService(  ) {
   }
   . . .
   . . .
   public HighLow_ServerSide getHighLow(String stock) {

      // retrieve the high and low for the specified stock
      return new HighLow_ServerSide((float)110.375,
                     (float)109.5);
   }
}
```

In order to make this new method available in Apache SOAP, we'll need to redeploy the service using a modified deployment descriptor. We need to add `getHighLow` to the list of methods, and add an entry in the mappings section for the high/low object. The second mapping entry defines the `HighLow` custom type, namespace-qualified using the service name `urn:BasicTradingService`. Here is the modified deployment descriptor:

```
<isd:service
    xmlns:isd="http://xml.apache.org/xml-soap/deployment"
    id="urn:BasicTradingService">
  <isd:provider
     type="java"
     scope="Application"
     methods="getTotalVolume getMostActive executeTrade executeStockTrade
           getHighLow">
    <isd:java
       class="javasoap.book.ch5.BasicTradingService"
       static="false"/>
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
```

```
  <isd:mappings>
    <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:BasicTradingService" qname="x:StockTrade"
       javaType="javasoap.book.ch5.StockTradeServer"
       java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
      xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    <isd:map
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       xmlns:x="urn:BasicTradingService" qname="x:HighLow"
       javaType="javasoap.book.ch5.HighLow_ServerSide"
       java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
      xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

Now we can create a client application that invokes the `getHighLow` service method and receives a high/low object in return. For the Apache SOAP client, we'll use the `HighLow_ClientSide` class to represent the return value. Here's the new application:

```
package javasoap.book.ch5;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.encoding.*;
import org.apache.soap.encoding.soapenc.*;
import org.apache.soap.util.xml.*;
public class HighLowClient
{
  public static void main(String[] args) throws Exception
  {
    URL url = new
        URL("http://georgetown:8080/soap/servlet/rpcrouter");
    Call call = new Call(  );
    SOAPMappingRegistry smr = new SOAPMappingRegistry(  );
    call.setTargetObjectURI("urn:BasicTradingService");
    call.setMethodName("getHighLow");
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
    call.setSOAPMappingRegistry(smr);
    BeanSerializer beanSer = new BeanSerializer(  );
    // Map the High/Low type
    smr.mapTypes(Constants.NS_URI_SOAP_ENC,
             new QName("urn:BasicTradingService", "HighLow"),
             HighLow_ClientSide.class, beanSer, beanSer);
    String stock = "XYZ";
    Vector params = new Vector(  );
    params.addElement(new Parameter("stock",
                            String.class, stock, null));
    call.setParams(params);
    Response resp;
    try {
      resp = call.invoke(url, "");
      Parameter ret = resp.getReturnValue(  );
      HighLow_ClientSide hilo =
              (HighLow_ClientSide)ret.getValue(  );
      System.out.println(hilo);
    }
```

```
      catch (SOAPException e) {
        System.err.println("Caught SOAPException (" +
                          e.getFaultCode( ) + "): " +
                          e.getMessage( ));
      }
   }
}
```

`smr.MapTypes( )` maps the `HighLow` custom type to the `HighLow_ClientSide` Java class. Just as before, we can use Apache's `BeanSerializer` to convert between XML and Java, since our class conforms to the JavaBeans property accessor pattern. We set up a single `String` parameter called `stock` to pass to the `getHighLow` method (although we don't actually make use of it in the server code). After the method is invoked, we cast the return value of `resp.getReturnValue( )` to an instance of `HighLow_ClientSide`. Then we pass the return parameter variable, `ret`, to the `System.out.println( )` method for display. This is all we need, since we implemented the `toString( )` method in our `HighLow_ClientSide` class. When you run this example, you'll get the following output:

```
High: 110.375 Low: 109.5
```

Here's the SOAP envelope returned from this method invocation. The `return` element is typed as a `HighLow` that is namespace-qualified by the `urnBasicTradingService` namespace. The properties, which are child elements of the `return` element, are typed as `float`s and appear along with their corresponding values.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <SOAP-ENV:Body>
    <ns1:getHighLowResponse
      xmlns:ns1="urn:BasicTradingService"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="ns1:HighLow">
         <low xsi:type="xsd:float">109.5</low>
         <high xsi:type="xsd:float">110.375</high>
      </return>
   </ns1:getHighLowResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

To return a `HighLow` using GLUE, we'll take the same steps we did for passing custom types. Again we'll separate our client-side work into another package, `javasoap.book.ch5.client`. Just restart the application of `BasicTradingApp` to get the service deployed. Now run the `wsdl2java` utility from the directory corresponding to the `javasoap.book.ch5.client` package:

```
wsdl2java http://georgetown:8004/glue/urn:BasicTradingService.wsdl
    -p javasoap.book.ch5.client
```

The following code is the new `IBasicTradingService` interface generated by `wsdl2java`. The `getHighLow( )` method returns an instance of `HighLow_ServerSide`; remember that this class is the new one generated by GLUE as part of the `javasoap.book.ch5.client` package, not the one being used by our server class, `BasicTradingApp`.

```
// generated by GLUE
package javasoap.book.ch5.client;
public interface IBasicTradingService
   {
   HighLow_ServerSide getHighLow( String arg0 );
   String executeStockTrade( StockTradeServer arg0 );
   String[] getMostActive(  );
   int getTotalVolume( String[] arg0 );
   String executeTrade( Object[] arg0 );
   }
```

The next class, `HighLow_ServerSide` , is simply a Java data structure reflecting the data fields that will be mapped to the `HighLow` custom type. I added the `toString( )` method by hand to make it simpler to display the results.

```
package javasoap.book.ch5.client;
public class HighLow_ServerSide {
   public float _high;
   public float _low;
   public String toString(  ) {
      return "High: " + _high +
         " Low: " + _low;
   }
}
```

Now let's create a client application using GLUE that invokes the `getHighLow` service method and displays the contents of the resulting return value. There's not much to this, really; we simply read the map file, perform the bind, and then call the `getHighLow` method on the bound interface. The resulting instance of `javasoap.book.ch5.client.HighLow_ServerSide` is passed to `System.out.println( )` for display. Just as in the Apache SOAP example, the `toString( )` method of the class handles the creation of the display string.

```
package javasoap.book.ch5.client;
import electric.registry.RegistryException;
import electric.registry.Registry;
import electric.xml.io.Mappings;
public class HighLowClient2 {
   public static void main(String[] args) throws Exception
   {
      try {
        Mappings.readMappings("BasicTradingService.map");
        IBasicTradingService srv = (IBasicTradingService)Registry.bind(
           "http://georgetown:8004/glue/urn:BasicTradingService.wsdl",
           IBasicTradingService.class);
        HighLow_ServerSide hilo = srv.getHighLow("ANY");
        System.out.println(hilo);
      }
      catch (RegistryException e)
      {
         System.out.println(e);
      }
   }
}
```

Here's the SOAP envelope returned from the server. You should be able to follow this by now. GLUE uses a reference to a separately serialized instance of the custom type, just as it did when returning an array.

```
<soap:Envelope
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <soap:Body>
      <n:getHighLowResponse xmlns:n='urn:BasicTradingService'>
        <Result href='#id0'/>
      </n:getHighLowResponse>
      <id0 id='id0' soapenc:root='0'
      xmlns:ns2='http://www.themindelectric.com/package/javasoap.book.ch5/'
        xsi:type='ns2:HighLow_ServerSide'>
        <_high xsi:type='xsd:float'>110.375</_high>
        <_low xsi:type='xsd:float'>109.5</_low>
      </id0>
    </soap:Body>
</soap:Envelope>
```

In this chapter, we've taken an in-depth look at the use of arrays and custom data types, and discussed how these structures are supported by Apache SOAP and GLUE. There are other useful types that you may find support for in these, or other, SOAP implementations. These types might include Java `Vector` and `Hashtable` classes, Java collections, and a variety of other commonly used Java classes.

We've seen that Apache SOAP and GLUE approach complex types in different ways, and both do a good job of supporting them. However, there may be times when you need to work with a custom type that either can't, or shouldn't, be serialized in the way provided by your SOAP implementation. This situation may arise because there simply is no support for a particular type of data, as is the case with multidimensional or sparse arrays, or perhaps for some other reason related to your application. We'll tackle this issue in the next chapter.